INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1011

Programme 1

A METHODOLOGICAL VIEW OF LOGIC PROGRAMMING WITH NEGATION

Pierre DERANSART
Gérard FERRAND

Avril 1989

PROGRAMME 1

UNE VUE MÉTHODOLOGIQUE DE LA PROGRAMMATION EN LOGIQUE AVEC NÉGATION

A METHODOLOGICAL VIEW OF LOGIC PROGRAMMING WITH NEGATION

Pierre DERANSART
INRIA
B.P. 105
78153 LE CHESNAY Cedex

Gérard FERRAND
LIFO - Université d'Orléans
Dept. Math. Informatique
B.P. 6759
45067 ORLEANS Cedex 2

Résumé :

Ce papier introduit des éléments de méthodologie de développement de logiciel dans le contexte de la Programmation en Logique, à savoir de programmes écrits en Clauses de Horn avec négation. On s'intéresse d'abord au problème de l'adéquation entre une sémantique attendue et celle, déclarative, attachée au programme. On donne alors des méthodes de preuve de correction et de complétude relativement à une sémantique attendue pour des programmes en clause définis comme des programmes avec négation. Dans ce dernier cas les méthodes présentées sont originales et aboutissent à des caractérisations simples de classes de programmes logiques connues dans la littérature. Il est suggéré que la simplicité apparente de la programmation en logique sans négation vient de la relative facilité des preuves et que la sémantique de la négation devrait également respecter cette simplicité.

Mots clés :

Programmation en Logique, Sémantique, Négation, Méthodes de preuve, Correction, Complétude.

Abstract :

This paper introduces some elements of a methodology of software development in the framework of Logic Programming, that is to say a way to write programs in the form of Horn Clauses with negation. It addresses the problem of the adequation between some intended semantics and some declarative one attached to the program. Proof methods for correctness and completeness of a logic program with regards to its intended semantics are presented for programs without and with negation in a common formalism. For programs with negation they are original ones. It is suggested that the apparent simplicity of logic programming (without negation) comes from the facility to handle such proofs and that the semantics of the negation should reflect this simplicity.

Keywords :

Logic Programming, Semantics, Negation, Proof method, Correctness, Completeness.

Recherche menée dans le cadre du groupe METHEOL du GRECO de Programmation PAOIA.

A METHODOLOGICAL VIEW OF LOGIC PROGRAMMING WITH NEGATION

Pierre DERANSART
INRIA
B.P. 105
78153 LE CHESNAY Cedex
FRANCE

Gérard FERRAND
LIFO - Université d'Orléans
Dept. Math. Informatique
B.P. 6759
45067 ORLEANS Cedex 2

1. Introduction

This paper introduces some elements of a methodology of software development in the framework of Logic Programming, that is to say Horn Clauses with Negation. It shows how the choice of a methodology may influence the choice of a semantics for the negation in Logic Programming.

In "Logic Programming" there is "Logic" and "Programming", or in other words programming using logic [Kow 74, 79]. According to common sense, and this corresponds in some extend to reality, "programming" is related with some operational view of the semantics of written programs, when "logic" claims that they have also a declarative semantics which can be understood in the framework of first order logic, without any operational view or any reference to some execution mechanism.

But it should be noted that there exists also a third element : the "intended semantics". This "intended semantics" corresponds to the intentions of the programmer. When it is formalized it corresponds to what can be called a specification. In Logic Programming it can be formalized by a set of atoms, i.e. a relation described by n-uples of values satisfying this relation. For example suppose we want to write a logic program to compute the length of a list. We may introduce two relations : the addition on integers (denoted "plus" with three arguments) and the length of a list (denoted "length" with two arguments) :

length (nil, zero).

length (A.L, N) ← length (L, M), plus (M, s(zero), N).

plus (zero, X, X).

plus (s(X), Y, s(Z)) ← plus (X, Y, Z).

No matter here whether this program is a "good" one. We focus our attention only on the "intention" which is : we want to specify (and furthermore to compute) the following sets of atoms :

for plus : plus (s^nzero), s^m(zero), s^{n+m}(zero)) n, m ≥ 0

and for length : length (t_1. .... t_n.nil, s^n(zero)) n ≥ 0, t_i terms.

In other words the third argument of "plus" is the sum of the two first if they are natural integers and the second argument of "length" is the length of its first argument which is a list of any length (lists of any elements).

It must be noted at this point that it is always possible, at least from a theoretical point of view, to represent the intended semantics by a set of well-formed atoms ([Cla 79, Hog 84, SS 86, Fer 87]). We use this approach for simplicity as it seems well adapted to describe the principles of the methodology we want to present herein. In practice some more convenient way may be used, but we do not develop this aspect in this paper. See for example [Cla 79, CD 88, DF 88a, Hog 84, DM 88, Dev 87] for different ways to specify the intended semantics or intended properties.

Coming back to the previous example the intended semantics looks clear — because the domains are well known — but now the questions are : does the given program fit with the intended semantics ? and what will happen when executing this program with some interpreter ? These two questions are related respectively with the **declarative semantics** and the **operational semantics**. Thus the process of software development should be analysed considering the three semantical levels : the intended semantics (IS), the declarative semantics (DS) and the operational semantics (OS). This is illustrated by the figure 1 :

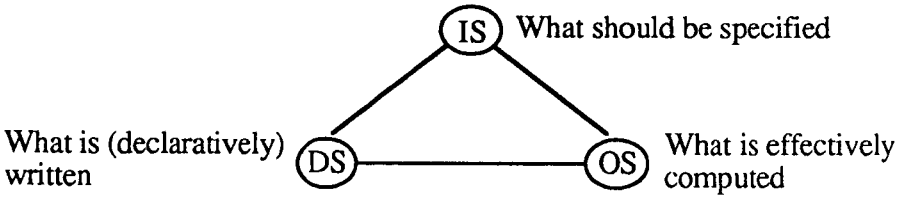

Figure 1 : the three semantics of a logic program

The purpose of **the methodology** is to describe the relationships between the three kinds of semantics. The introduction of the intended semantics is something new. In fact it is generally agreed that the declarative semantics of logic programs is clear enough in such a way that it reflects the intended semantics. In other words the two ways (declarative and operational) to look at a logic program permit to consider it as a specification or a program as well. This view does not correspond exactly to the practice. A program with a clear declarative semantics cannot always be executable for some goals. Thus the programmer is often encouraged to give more attention to what will be effectively computed, forgetting the declarative view. This comes from the fact that logic programming is still a low level style of programming.

The given example will illustrate this point. With a standard interpreter, as it is written, this program will work well for goals whose first argument is a given list. But it does not terminate if the first argument of "length" is a variable. One may observe that the way the second clause is written reflects some intended use coming from an operational view. For the "reversed" use one should write :

length (A.L, N) ← plus (M, s(zero), N), length (L, M).
as plus terminates if its first or last argument is ground

Now the third equivalent version like :

length (A.L, N) ← plus (s(zero), M, N), length (L, M).
will work well with every goal "length" with one of the arguments instantiated.

All these programs are obviously equivalent from the declarative semantics point of view. But what can be observed is that with such writing methodology we are considering two problems at the same time : the relationships between DS and IS (correctness of the program from the point of view of the intended semantics) and the relationships between OS and IS.

Our suggestion is that both problems should be analysed separately (at least from a methodological point of view) following the scheme of figure 2 :

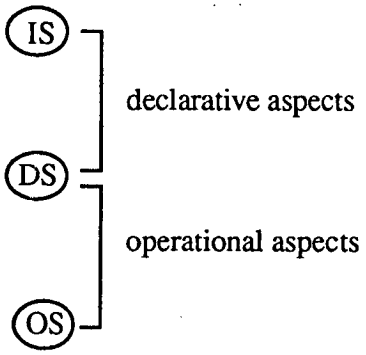

Figure 2 : suggested methodology

One should analyse first the relationships between IS and DS showing that they are equal, hence giving first attention to the declarative view. It is the **declarative step**. Then one should care about execution, trying to preserve as much as possible the same semantics through computations (**operational step**). Depending on the interpreters it is not always possible to have only one executable program for any kind of goals.

As these steps are obviously related (if some part of the semantics is not computable with some interpreter for a given program, one may modify the intended semantics as the program as well), they should be treated in a common framework. Our suggestion is to use the notion of **proof-tree** as the basic notion which permits to relate the three levels of semantics. More precisely the two steps are thus the following :

- **The declarative analysis step** in which relationships between IS and DS are considered. For programs without negation the notion of proof-tree formalizes the notion of proof of atomic theorems ([Cla 79, DM 85]). In the case of negation the notion of proof-tree can be generalized [DF 87a, 88c]. Thus the declarative semantics is the set of all the proof-tree roots. Hence the adequation between IS and DS consists in showing **the correctness** (DS $\subseteq$ IS), i.e. that all the proof-tree roots are in IS, and **the completeness** (IS $\subseteq$ DS), i.e. that all the intended atoms are proof-tree roots. The notion of proof-tree permits to make simple foundations of proof methods, essentially based on inductions on proof-trees.
- **The operational analysis step** in which the operational properties of the program are established in comparison with DS. The operational semantics describes the way to compute proof-trees. Depending on the interpreters and the proof-tree building strategies they include, not all proof-trees are always possible to compute (loop may appear). In that case the operational semantics will not be complete in the sense of DS (hence IS, as the identity of IS and DS has been established previously).

Many operational properties may be proven like run-time properties in [DrM 87, DM 88, Der 88] or termination, but the main purpose of the methodology consists in finding or verifying the conditions such that the operational semantics remains correct and complete with regard to DS.

In principle, one should make sure that the correctness will be preserved and, if the completeness cannot be reached, one should be able to characterize the successful computations.

As illustration consider the following program which specifies the inclusion of two sets represented by lists of elements (repetitions are possible) :

include (L_1, L_2) $\leftarrow$ not ninclude (L_1, L_2)

ninclude (L_1, L_2) $\leftarrow$ elem (X, L_1), not elem (X, L2)

elem (X, X.L)

elem (X, Y.L) $\leftarrow$ elem (X, L).

In the first step the methodology will give proof methods showing that this program is correct and complete with regard to the intended semantics :

include (L_1, L_2) : L_1 et L_2 are ground lists such that $L_1 \subseteq L_2$.

In the second step the methodology will help to be sure that by using some implementation of the negation, the inclusion of two given ground lists will be computed successfully if they are included, and with failure if not. For example for standard interpreters implementing SLDNF-resolution, it will be sufficient to show that the negation is "safely" used [Llo 87].

The methodology we present in that paper concerns the first step only. It addresses the problem of the adequation between the intended and the declarative semantics. For programs

without negations some solutions have been already proposed [Cla 79, Hog 84]. We recall here some other solutions based on the notion of proof-tree as they have been already proposed in [CD 88, Der 84, DF 87a, b], but with a different formalism. Then we present a new solution for programs with negation, keeping the same approach based on the notion of (generalized) proof-tree. This preserves the simplicity of the proof methods, hence the simplicity of the declarative semantics. As a consequence the kind of negation we are dealing with is slightly different from others which have given rise to many publications until now. But these differences, if compared with the closed word assumption, or the completion of axioms [AvE 82] are minor and this negation is very close to that of [ABW 86, PP 88]. However it should be noticed that the purpose of this paper is not to study the semantics of this new negation (such a study is in DF 87a, 88c), rather to present the basic motivations of such kind of negation based on methodological concerns.

This paper does not develop the methodology attached to the second (operational) step. It has been in [DF 88a, b]. But the relationships between DS and OS have been extensively studied until now. Usually DS is the logical semantics represented by the completion of the axioms in case of negation ([Cla 79, AvE 82, Llo 87], see [Del 86] for a study of different semantics) and OS is some resolution procedure, for example SLD or SLDNF-resolution ([Llo 87, DF 87a, WML 84, Der 88] and many others).

This paper is organized into two parts : methodology of logic programming without negation (section 2) and with negation (section 3). Proof methods of completeness and correctness with regards to a specification (the intended semantics) are presented in both cases with proofs of soundness and completeness of the methods and illustrating examples.

2. Elements of Methodology of Logic Programming without negation

We borrow from [Cla 79, DM 85, DF 87b] some well-known concepts.

2.1. Abstract Syntax (Terms, Atoms, Clauses)

- Terms : They are built as usually with variables, constants and functors. Terms will be untyped. This convention will oblige us to introduce in logic programs some typing predicates. For example integers will be specified by a predicate defined by clauses :

  int (zero)
  int (sX) ← int (X).

  or lists :

  list (nil)
  list (A.X) ← list (X).

  Well-formed or typed terms could be used as in [Der 89] but with untyped terms the theory as presented here is simpler, without lost of generality. *A ground term* is a term without variable.

- Atoms (atomic formulæ) built with predicate symbols and terms. We denote by ATOM the set of all the (non necessarily ground) atoms.

- Clauses $a_0 \leftarrow a_1, \ldots, a_n$. $n \geq 0$ where a_0 is the head, $a_1, \ldots, a_n$ the body. If $n = 0$, a_0 is a fact.

- Program : a set of clauses grouped into packets.

- Goals : in general a body limited to one atom.

Note that the notions of substitution and instances apply as usual to terms, atoms, goals or clauses.

2.2. Declarative Semantics (Proof trees)

The declarative semantics DS is the set of the proof-tree roots. A proof-tree ([Cla 79, DM 85]) is a tree built with instances of clauses, whose leaves are instances of facts. Note that by definition the proof-trees are not necessarily ground and that any instance of a proof-tree is a proof-tree. It follows that DS is closed by substitution. (DS is denoted DEN(P) in [DF 87a, b] and called the denotation or constructive semantics).

2.3. Note on the Logical Semantics

As proof-trees are representation of a proof of its root, proof-tree roots are atomic theorems and DS is also the set of all the (atomic) logical consequences of the axioms. It is also the least (not ground) Herbrand model of the program [Cla 79, Fer 87].

2.4. Intended Semantics, Correctness, Completeness of a program.

The Intended Semantics (IS) is a set of atoms which is the disjoint union of a family of sets $\{IS^p\}_{p \in PRED}$, such that $IS^p \subseteq \{p(t_1, \ldots, t_n) \mid n \text{ arity of } p \text{ and the } t_i\text{'s are well-formed terms}\}$. Thus $IS = \bigcup_{p \in PRED} IS^p$.

IS is supposed to be closed by (well-formed) substitutions.

The aim of the methodology is to prove that DS and IS are identical (i.e. DS = IS).

A program P is said **(partially) correct w.r.t. IS** iff for all atom a in DS, a s in IS, i.e. :

$$DS \subseteq IS.$$

A program P is said **compete w.r.t. IS** iff for all atom a in IS, it exists a proof-tree whose root is a, i.e. :

$$IS \subseteq DS.$$

2.5. Proof method for correctness

In [CD 88, Der 89] and, with a closer formalism in [DF 87], the following **method** is proven sound and complete.

Find some subset IS' $\subseteq$ IS closed by substitution such that
For every clause C : $a_0 \leftarrow a_1, \ldots, a_n$ the following statement holds :
$\forall\sigma$ (substitution) $[(\forall i, 1 \leq i \leq n, \sigma a_i \in IS') \Rightarrow \sigma a_0 \in IS']$

It is clear that such a method is sound as every proof tree is built with well-formed clauses instances and such implications insure that every proof tree root will belong to IS' hence to IS, by hypothesis. The converse holds trivially with IS' = DS, as, by hypothesis, DS $\subseteq$ IS hence the method is complete (see [CD 88 or Der 89] for a detailed discussion about this point).

This proof method is extremely simple. It can be viewed from a practical point of view as a way to get convinced that the program is built with axioms (by the way, implications (1) are the definition of the fact that IS' is a (termal) model of the axioms — see [Fer 87 or DF 87a] for more details). But its simplicity suffers some drawbacks, as noticed in [CD 88], coming from the simplicity of the definition of the intended semantics. Thus other and more tractable proof methods have been developed in [CD 88, Der 84, 89] for the same purpose. We do not develop such methods herein. A short example will help to understand the general and simple method presented here :

P : C1 : plus (zero, X, X) $\leftarrow$ int (X).
C2 : plus (sX, Y, sZ) $\leftarrow$ plus (X, Y, Z)

IS^{plus} : plus (s^n zero, s^m zero, s^{n+m} zero), $n \geq 0$, $m \geq 0$
int as in 2.1
IS^{int} : int (s^nzero), $n \geq 0$

C1 : $\forall\sigma\ \sigma$int(X) $\in IS^{int} \Rightarrow \sigma$plus (zero, X, X) $\in IS^{plus}$
C2 : $\forall\sigma\ \sigma$plus (X, Y, Z) $\in IS^{plus} \Rightarrow \sigma$plus (sX, Y, sZ) $\in IS^{plus}$
obvious as for the clauses of int.

In 2.7. some complete examples are developed.

2.6. Proof method for completeness

Joining results obtained in [CD 88 and Der 84], also described with a closer formalism in [DF 87a, b] we get the following **method** which we show sound and complete :

Find some superset IS' of IS (i.e. $IS \subseteq IS'$) closed by substitution and a function f such that :

a) For every predicate p, IS'^p is the (not necessarily disjoint) union of subsets IS'^p_k, each of them being associated to one clause whose head is an atom of p. In other words to every clause c^p_k of the packet of p (denoted C^p) is associated a subset IS'^p_k such that :

$$IS'^p = \bigcup_k IS'^p_k$$

The sets IS'^p_k are assumed closed by substitution.

b) For every predicate p and for every clause $c^p_k : a_0 \leftarrow a_1, \ldots, a_n$ of the packet C^p the following statement holds :

$\forall a \in IS'^p_k \quad \exists \sigma : \sigma a_0 = a$ and

b1) $\forall i, 1 \leq i \leq n \quad \sigma a_i \in IS'$ and

b2) for some j's $\quad f(\sigma a_j) < f(\sigma a_0)$

in which in the conjunction "for some j" denotes the atoms which may correspond to "recursive calls" of the predicate p (directly or not) and f is a function from atoms into a set in which "<" denotes a well-founded relation (there is no infinite decreasing sequence in it).

In other words (soundness of the method) conditions a and b1 express that for every atom in IS^p it is possible to find an instance of a clause (by σ) in the packet of p such that the head instance is this atom and the instances of the atoms in the body are in IS. Thus it is possible to continue the "construction". Notice that this kind of "construction" is not the operational semantics with unification. It consists only in pasting together instances of clauses. The criterion b2 ensures that such a "construction" is finite, which means (with criteria a and b1) that it will be achieved with success. Hence the existence of a proof tree for every atom in IS. The converse is not difficult to prove with IS' = DS and using as function f the height of the proof-trees (completeness of the method).

The same example will illustrate the method.

P : C1 : plus (zero, X, X) $\leftarrow$ Int (X).
C2 : plus (sX, Y, sZ) $\leftarrow$ plus (X, Y, Z)

$IS^{plus} = plus (s^n zero, s^m zero, s^{n+m} zero), n \geq 0, m \geq 0$

a) $IS^{plus}_1 = plus (zero, s^m zero, s^m zero), m \geq 0$
$IS^{plus}_2 = plus (s^n zero, s^m zero, s^{n+m} zero), n > 0, m \geq 0$
$IS^{plus} = IS^{plus}_1 \cup IS^{plus}_2$

b) choose $f(plus(s^n zero, t_1, t_2)) = n$
In C1 : $\sigma : X \leftarrow s^m zero$, hence $int(s^m zero)$ belongs obviously to IS^{int}
clause 2 is the only possiblity of recursive call.
In C2 : (Note σ is ground)
$\forall \sigma \; \sigma$ plus (sX, Y, sZ) $\in IS^{plus}_2$

$\Rightarrow$ (b1) σ plus (X, Y, Z) $\in$ IS^{plus} and

(b2) f (σ plus (X, Y, Z)) < f (σ plus (sX, Y, sZ))

Obviously true as σ is, by hypothesis :

$X \rightarrow s^n zero$, $Y \rightarrow s^n zero$, $Z \rightarrow s^{n+m} zero$ $\quad n \geq 0, m \geq 0$

and plus ($s^n zero$, $s^n zero$, $s^{n+m} zero$) belongs to IS^{plus}, hence the results

with b2 : $n < n + 1$.

"int" is obviously complete with regards to IS^{int}.

As quoted for the (partial) correctness method, this method suffers from being too simple in many practical situations. Its advantage is to formalize also the simplicity of logic programming : each clause in a packet may correspond to some (non exclusive) case such that — at some intuitive level — the head is equivalent to the disjunction of the bodies. This kind of idea is also formalized in the notion of completeness [AvE 82] or of "logic algorithm" in [Dev 87] but in these approaches this semantics denotes some "belief" as quoted in the introduction, when here it is clearly related to the existence of a proof tree for every atom in the intended semantics (This does not mean that such proof trees will be computable with any interpreter — which is a problem of relationships with the operational semantics).

However the (apparent) simplicity of the proof method is related to the apparent simplicity of logic programming such that the programmer can be easily convinced that he is writing correct and complete axiomatization of some intended semantics (or also model).

2.7. Examples

We achieve firstly the previous example and give one other.

2.7.1. Coming back to the example of the introduction, we have :

- IS^{plus} = plus ($s^n zero$, $s^m zero$, $s^{n+m} zero$), $n \geq 0, m \geq 0$
  IS^{length} = length ($t_1.t_2$, ... t_n.nil, $s^n zero$), $n \geq 0$, t_i denotes any term.
- P : C1 plus (zero, X, X) $\leftarrow$ int (X)
  C2 plus (sX, Y, sZ) $\leftarrow$ plus (X, Y, Z).
  C3 length (nil, zero).
  C4 length (A.L, M) $\leftarrow$ plus (N, zero, M), length (L, N).

Correctness and completeness of plus and int already proven w.r.t. $IS^{plus} \cup IS^{int}$

Correctness of P w.r.t. IS (= $IS^{length} \cup IS^{plus} \cup IS^{int}$) :

C3 : $\forall\sigma$(empty) length (nil, zero) $\in$ IS^{length}. Obvious (n = 0).

C4 : $\forall\sigma(A \rightarrow t, L \rightarrow t_1. \ldots.t_n.nil, M \rightarrow s^{n+1} zero, N \rightarrow s^n zero), n \geq 0$

σplus (N, szero, M) $\in$ IS^{plus} $\wedge$ σlength (L, N) $\in$ IS^{length} $\Rightarrow$ σ length(A.L, M) $\in$ IS^{length}

: σlength (A.L, M) = length($t.t_1 \ldots t_n$.nil, $s^{n+1} zero$) $\in$ IS^{length}

Completeness of P w.r.t. IS (= $IS^{length} \cup IS^{plus}$) :

a) $IS^{length} = IS^{length}_3 \cup IS^{length}_4$ with

IS^{length}_3 = length (nil, zero)

IS^{length}_4 = length (t_1. .... t_n.nil, s^nzero) $t_i \in$ term, $n > 0$

b) Choose the following function :

f(length(t_1. .... t_n.nil, t)) = n

f(plus(s^nzero, t_1, t_2)) = n

only clause C4 has to be considered (clause C2 has already been) for decreasing f :

$\forall a \in IS^{length}_4$

$\exists \sigma$(A → t_1, L → t_2. ....t_n.nil, M → s^nzero, N → s^{n-1}zero) σ length (A.L, M) = a and

(b1) σ plus (N, szero, M) $\in IS^{plus} \wedge \sigma$ length (L, N) $\in IS^{length}$

and (b2) f(σ length(L, N)) < f(σ length(A.L, M))

hence the results for the atom length in the body and :

f(length(t_1. .... t_n.nil, t') < f(length(t.t_1. .... t_n.nil, t").

Note that no decreasing is necessary for atom plus in the body of C4 as no recursion is possible through this atom.

The case of clause C3 (with IS^{length}_4 = length(nil, zero)) is trivial.

2.7.2 Here is an other example for which a proof seems necessary in practice to be convinced of its correctness : the palindrom program with difference list [CT 77] :

IS^{palin} = palin (t_1. .... t_n.l — l) $n \geq 0$, l is a list as specified by predicate "list" in 2.1.

and t_1 .... t_n.nil = reverse (t_1 .... t_n.nil)

IS^{list} = list (t_1..... t_m.nil), $m \geq 0$.

Palin = C1 : palin (L — L) ← list (L).

C2 : palin (A.L — L) ← list (L).

C3 : palin (A.L — M) ← palin (L — A.M).

(Partial) Correctness with regards to IS^{list} = list (u_1. ... .u_m.nil), $m \geq 0$, u_i terms, union IS^{palin} :

C1 : $\forall\sigma$(L → u_1. ... .u_m.nil) σlist) (L) ⇒ σpalin (L — L) $\in IS^{palin}$

(n = 0 and nil = reverse (nil))

C2 : also obvious.

C3 : $\forall\sigma$(A → t, L → t_1 .... t_n.t.l, M → t.l) with t_1. .... t_n.nil = reverse (t_1. ....t_n.nil)

σ palin (L — A.M) $\in IS^{palin}$ ⇒ σ palin (A.L — M) $\in IS^{palin}$

as $t.t_1$. .... t_n.t.nil = reverse (t.t_1. ....t_n.t.nil)

The properties of "reverse" are supposed to be known.

Completeness :

a) $IS^{palin} = IS^{palin}_1 \cup IS^{palin}_2 \cup IS^{palin}_3$

with IS^{palin}_1 = palin (l — l) l list ($l = u_1. \ldots u_m.nil$, $m \geq 0$)

IS^{palin}_2 = palin (t.l — l) t term, l list

IS^{palin}_3 = palin ($t.t_1. \ldots t_n.t.l$ — l) $n \geq 0$, t, t_i terms l list.

and $t_1. \ldots t_n.nil$ = reverse ($t_1. \ldots t_n.nil$)

b) we choose :

f(palin($t_1. \ldots t_n.l$ — l)) = n + m where m is the length of the list l ($l = u_1. \ldots u_m.nil$, $m \geq 0$)

The only difficulty is to prove : $\forall a \in IS^{palin}_3$,

$\exists \sigma(A \rightarrow t, L \rightarrow t_1. \ldots t_n.t.l, n \geq 0, l = u_1. \ldots u_m.nil, m \geq 0, M \rightarrow l)$

σ palin (A.L — M) = a and

(b1) σ palin (L — A.M) $\in IS^{palin}$ and

(b2) f(σ palin (L — A.M)) < f(σ palin (A.L — M))

Obvious as if reverse ($t.t_1. \ldots t_n.t.nil$) = $t.t_1. \ldots t_n.t$ nil

then reverse ($t_1. \ldots t_n.nil$) = $t_1. \ldots t_n.nil$.

hence the results :

palin ($t_1. \ldots t_n.t.l$ — t.l) $\in IS^{palin}$ and

f(palin ($t_1. \ldots t_n.t.l$ — t.l)) < f (palin($t.t_1, \ldots t_n.t.l$ — l)).

(as n + 1 + m < n + 2 + m).

Remark that the clause C2 is needed as the proof for C3 uses the hypothesis that $t.t_1. \ldots .t_n.t.l$ — l represents a list with at least two elements. Moreover one could observe that from an operational point of view it could be sufficient to write a program with IS^{palin} = palin ($t_1. \ldots t_n.nil$ — nil) as (as the proof of completeness of this deterministic program shows) only goals having finite lists in the argument will terminate. Thus one could expect that clauses C1 and C2 are too general ones. Unfortunately proof in clause C3 would not be possible any more with such IS, hence variables in C1 and C2 are necessary to ensure completeness.

3. Elements of Methodology of Logic Programming with Negation

3.1. Abstract syntax

We consider now normal programs [Llo 87] in which the clauses have the following syntax :

$$a_0 \leftarrow l_1, \ldots, l_n, n \geq 0$$

where l_i is a_i or (not a_i).

All other notions are unchanged.

3.2. Declarative semantics (proof-tree with negation)

We present a semantics of the negation based on an extended notion of proof-tree. Its advantage is to permit a uniform treatment of the methodology (proofs of identity of IS and DS and relationships with OS), but it differs from other semantics of the negation. This point will be discussed below.

Definition 3.2.1. : Semi-Proof-Tree (SPT)

A Semi-Proof-Tree (SPT in short) is a proof-tree (section 2.2) with the difference that there is no restriction on the negative leaves.

This definition keeps the idea of proof-tree built with instances of clauses. But the leaves are positive or negative ones. Positive leaves are instances of facts. The idea of negative leaves (not a) corresponds to atoms a which are never true, hence a cannot be a proof-tree root. This idea leads to a paradox : take for example the program :

$$p \Leftarrow \text{not } p$$

thus p is a proof-tree root iff p is **not** a proof-tree root. the following definitions are stated with the purpose to overcome this paradox.

Definition 3.2.2. : Set of useful atoms U, semi-proof tree on U (SPT on U)

We consider a subset U of ATOM ($U \subseteq ATOM$) called the set of **useful atoms** which is closed by substitution. Note that ATOM and ATOMG (ground atoms) are possible candidates.

A **semi-proof-tree on U** is a semi-proof-tree in which all nodes a or (not a) satisfy $a \in U$.

Definition 3.2.3. : (Relation on SPT on U : >, well-foundness)

We denote by > a relation defined on the set of the SPT on U as follows :

T > T' iff there exists a negative leaf of T, (not a), such that root (T') is an instance of a.

$(\exists \sigma, \sigma a = \text{root } (T'))$

The relation > is **well-founded** iff there does not exist infinite "decreasing" sequence $a_0 > a_1 > \ldots$. (usually > is not an order).

The idea of the relation is to characterize some way to combine proof-trees, exhibiting eventual paradoxes. With the given example and definition 3.2.3., the unique semi-proof-tree $t : p \leftarrow \text{not } p$ satisfies $t > t > \ldots$

Thus the fundamental theorem which ensures the existence of a semantics based on the notion of semi-proof-tree .

Theorem 3.2.4. : Proof-Tree with Negation (PTN)

If the relation > (definition 3.2.3.) is well-founded then there exists a **unique** set of SPT on U, denoted by PTN, which satisfies the condition :

For all semi-proof-tree on U, T :

$T \in PTN \Leftrightarrow$ For every negative leave (not a) of T no instance of a is the root of a semi-proof-tree of PTN.

Sketch of the proof : the condition is equivalent to a definition by induction on the well-founded relation.

To be more precise :

a) Let SPT(U) be the set of the SPT on U. We have to prove that there exists a unique function

$f : SPT(U) \rightarrow \{0, 1\}$ verifying the condition :

For all $T \in SPT(U)$,

$f(T) = 1 \Leftrightarrow$ for every negative leave (not a) of T no instance of a is the root of a $T' \in SPT(U)$ such that $f(T') = 1$.

But in the second member we can add $T > T'$, so this condition is equivalent to

$f(T) = 1 \Leftrightarrow$ for every $T' \in SPT(U)$ such that $T > T'$, for every negative leave (not a) of T, if $f(T') = 1$ then no instance of a is the root of T'.

b) Now we are going to use the principle of definition by induction on a well-founded relation. We recall here the principle (for more details, see for instance [Acz 77]) :

Let W be a set and > a binary relation well-founded on W.

Let E be a set. For each $w \in W$, we call w-family any family of elements of E whose index set is $\{w' \in W \mid w > w'\}$.

Let WF the set of all the w-family, for all $w \in W$.

The principle says that, for each function

$$F : WF \rightarrow E$$

there exists a unique function $f : W \rightarrow E$ such that, for all $w \in W$,

$$f(w) = F((f(w'))_{w > w'})$$

c) We get our result with :

$W = SPT(U)$

$E = \{0, 1\}$

$F((e_{T'})_{T > T'}) = 1$ if for every negative leave (not a) of T,
for every $T' < T$ such that $e_{T'} = 1$,
no instance of a is the root of T'.
$= 0$ otherwise QED.

Definition 3.2.5. : Declarative Semantics (DS_U, neg_U)

The elements of PTN will be called the proof-trees (with negation) on U.

Again not all proof-trees are always possible to compute : at first for the same reason as without negation (introduction, the operational analysis step and section 2.1), and also because of the implementation of the negation. The programmer must certify that only semi-proof-trees on U will be computed. This point is related to the operational semantics and is not developed here.

The **declarative semantics** of P on U (denoted DS_U) is the set of all the proof-tree roots on U.

We denote neg_U the set of the atoms in U such that they don't have any instance in DS_U.

We give now some sufficient condition such that the proof-trees on U are well-founded (i.e. the relation > is well-founded) :

Property 3.2.6. : Condition of well-foundness on U

The relation > (def. 3.2.3.) is well founded if the following condition holds :
It exists a function $g : U \rightarrow \mathbb{N}$ (natural integer) such that :
1) $\forall$ a in U, $\forall \sigma : g(\sigma a) \leq g(a)$
2) In every semi-proof-tree on U of root a, all negative leaves (not b) satisfy $g(b) < g(a)$.

Proof : It is sufficient to show that if $T > T'$ then $g(root(T)) > g(root(T'))$ holds. By (C2) $g(root(T)) > g(b)$ holds for the atoms b in every negative leaves and by (C1) $g(b) \geq g(root(T'))$. QED.

Remark : this condition may appear very sophisticated. But it is satisfied if the program P is stratified in the sense of [ABW 86] for any U. A program is stratified if no recursion goes through the negation. For ground U the conditions are analogous to that of [PP 88].

Notations : Let I be a set of atoms. We will denote by neg(I) the set of atoms in U which do not have any instance in I. Note that $neg_U = neg(DS_U)$, and neg(I) is closed by substitution.

Now we can give a characterization of the declarative semantics of logic programs with negation.

Corollary 3.2.7.

DS_U is the **unique** subset I of U which satisfies :

$a \in I \Leftrightarrow$ a is the root of a semi-proof tree on U whose negative leaves (not b) satisfy $b \in neg(I)$.

Proof (Existence) (Unicity). By definition 3.2.5. DS_U satisfies the property. If I satisfies the property then the set of semi-proof-trees whose negative leaves are in neg(I) satisfies the theorem 3.2.4. QED.

3.3. Methodology

The purpose of the methodology is to prove $IS = DS_U$. This will be done by proving together correctness and completeness. Thus we have to verify (by definition of the methodology, and corollary 3.2.7.).

$a \in IS \Leftrightarrow$ a is the root of a semi-proof tree on U whose negative leaves (not b) satisfy $b \in neg(IS)$.

This can be achieved by satisfying the three following independent steps :

[1] The relation > (for P and U) is well-founded (if P is stratified this condition holds for any U).

[2] Give the sets neg(IS) from IS. (Note that neg(I) is a function of I).

[3] Prove the following conditions :

let as in 2.6 (a) $IS = \cup IS^P$ and $IS^P = \cup IS^P_k$ and f be some function, in every clause $C : a_0 \leftarrow l_1 \ldots l_n$, the following statements hold :

1) $\forall\sigma\ [(\forall i, 1 \le i \le n\ \ \sigma l_i \in (IS \cup Nneg(IS))) \Rightarrow \sigma a_0 \in IS]$
where $(not\ a) \in Nneg(X)$ iff $a \in neg(X)$

2) b) $\forall a \in IS^P_k \quad \exists\sigma \quad \sigma a_0 = a$ and
b1) $\forall i, 1 \le i \le n \quad \sigma l_i \in IS \cup Nneg(IS)$ and
b2) for some j's $\quad f(\sigma a_0) > f(\sigma a_j)$

Theorem 3.3.1. (soundness and completeness of the method)

Under the hypothesis of condition [1], the program P on U is correct and complete with regards to IS, i.e. $IS = DS_U$ iff the conditions [3] hold.

Proof : conditions are the strict translation of the definition of the methodology $a \in IS \Leftrightarrow$ there exists a semi-proof-tree on U whose negative leaves (not b) are in Nneg(IS) and same reasoning as in proof of section 2.6 can be used for f. Thus IS is the unique set satisfying corollary 3.2.7. QED.

Remark : One should remark the analogy between the conditions [3] in the methodology with negation and the conditions of correctness an completeness in the methodology without negation.

If there is no negation we find the same conditions as in section 2.5 and 2.6 (take DS_U in state of DS). This shows that the kind of negation considered here keeps the simplicity of the axiomatic style of logic programming. However one could be tempted to split conditions [3] such that each part would correspond to one inclusion : [3](1) to $DS_U \subseteq IS$ and [3](2) to $DS_U \supseteq IS$. This is not the case since [3](1) $\Rightarrow DS_U \subseteq IS$ holds if we assume completeness ($IS \subseteq DS_U$). This can be achieved if one manages the proof level by level (see theoretical and practical description in [DF 87a]).

3.4. Examples

3.4.1. Inclusion of two sets

Program :

1 - included (L_1, L_2) ← not nincluded (L_1, L_2), glist (L_1), glist (L_2).
2 - nincluded (L_1, L_2) ← elem (X, L_1), not elem (X, L_2), glist (L_1), glist (L_2), gelem (X).
3 - elem (X, X.L) ← glist (L), gelem (X).
4 - elem (X, Y.L) ← elem (X, L), gelem (Y).
5 - glist (nil).
6 - glist (X.L) ← glist (L), gelem (X).
7 - gelem (a).
. .
. .

in which "gelem" defines possible ground elements of a list and "glist" defines ground lists as in 2.1.

Conditions :

[1] The program is stratified.

[2]

	IS	neg(IS)
included	ground L_1, L_2, $L_1 \subseteq L_2$	not needed
nincluded	ground L_1, L_2, $L_1 \not\subseteq L_2$	ground L_1, L_2, $L_1 \subseteq L_2$
elem	ground X, L, $X \in L$	ground X, L, $X \notin L$
glist	ground L	not needed
gelem	ground X	not needed

[3] Conditions trivially hold for clauses 1 and 2.

For elem (clauses 3 and 4) correctness and completeness are proven without negation.

3.4.2. Example (analogous to [PP88])

Program : 1 - even (zero).
2 - even (X) ← is_suc (X, Y), not even (Y)
3 - is_suc (sX, X) ← int (X).

Conditions :

[1] U = ground atoms built with Int

> is well-founded as conditions 3.2.6 hold :

choose : f(even(s^nzero)) = n

[2]

	IS		neg(IS)	
even	even(s^{2n}zero)	$n \geq 0$	s^{2n+1}zero	$n \geq 0$
is_suc	is_suc(s^{n+1}zero,s^nzero)	$n \geq 0$	not needed	
int	int(s^nzero)	$n \geq 0$	not needed	

[3] 1) even(zero) $\in s^{2n}$(zero) (n = 0)

and $\forall\sigma$ (X $\rightarrow s^p$zero, Y $\rightarrow s^q$zero)

is_suc(σX, σY) $\in$ is_suc(s^{n+1}zero, s^nzero) $\wedge$ even(σY) $\in$ even(s^{2n+1}zero)

$\Rightarrow$ even(σX) $\in$ $s2^n$zero

OK with p = 2n' + 2, q =2n' + 1 and n = n' + 1

2) a) IS^{even}_1 = even(zero)

IS^{even}_2 = even(s^{2n}zero) n > 0

b1) same kind of reasoning, but implication reversed and existence of σ guaranteed by the head even(X) in which the term is a variable.

b2) use : f(even(s^nzero)) = n.

4. Discussion, conclusion

Proof method of correctness and completeness with regard to a specification have been presented and proven sound and complete. They seem very natural on a practical point of view. The concepts of correctness and completeness are globally equivalent to the same concepts introduced in [Cla 79, Hog 84, KS 86, Tär 86, Dev 87, SS 86]. Only in [Dev 87] the idea of a proof method for correctness and completeness is presented for programs with negation but it is based as usual on induction on the intended semantics defined as the unique model of some completion of the axioms. In our approach the axioms are used without any modification, keeping there original form and there is no restriction a priori on the models.

We have presented a methodology of normal logic program development in which declarative and operational properties are considered separately. The main advantage of such a methodology is to permit to the programmer to focus his attention to the declarative semantics of its programs. As a consequence the semantics of the negation is slightly different from others already known [AvE 82, ABW 86, PP 88, Kun 87], but it coincides in most of the practical cases with the usual ones. Some theoretical aspects of such a negation have been developed in [DF 87a]. Its drawbacks is that no simple logical semantics is always possible but its main advantage is to preserve the simplicity of the declarative semantics by keeping the approach based on the proof-trees still valid. It follows that operational aspects can be analysed easily in the same framework and that the logic program developed by that way can be more easely made runnable.

Bibliography

[ABW 86] Apt K.R., Blair H., Walker A. : Toward a theory of declarative knowledge. Foundation of Deductive Databases and Logic Programming. Minker J., (ed) Morgan Kaufman, Los Altos 1987.

[Acz 77] Aczel P. : An introduction to inductive definitions, in Handbook of Math. Logic, Barwise (Ed), North Holland 1977, pp. 739-782.

[AvE 82] Apt K. R., Van Emden M. H. : Contributions to the theory of Logic Programming. JACM, 29, 3, pp 841-862, July 1982.

[CD 88] Courcelle B., Deransart P. : Proof of Partial Correctness for Attribute Grammars with application to Recursive Procedure and Logic Programming. Information and Computation 78, 1, July 1988.

[Cla 79] Clark K.L. : Predicate logic as a computational formalism. Res. Rep., Dept of Computing, Imperial College, London, 1979.

[CT 87] Clark K. L., Tärnlund S. A. : A first Order Theory of Data and Programs. IFIP 87, North Holland, pp 939-944, 1987.

[Del 86] Delahaye J. P. : Sémantique logique et dénotationnelle des interpréteurs PROLOG. Note IT n° 84, University of Lille, 1986.

[Der 84] Deransart P. : Validation des Grammaires d'Attributs. Doctoral Dissertation N°822, University of Bordeaux I, October 1984.

[Der 88] Deransart P. : On the Multiplicity of operational Semantics for Logic Programming and their Modelization by Attribute Grammars. INRIA - RR 916, October 1988.

[Der 89] Deransart P. : Proofs of Declarative Properties of Logic Programs. Tapsoft'89, Barcelone, March 13-17, 1989.

[Dev 87] Deville Y. : A Methodology for Logic Program Construction. PhD Thesis, Institut d'information, Facultés Universitaires de Namur (Belgique), February 1987.

[DF 87a] Deransart P., Ferrand G. : Programmation en logique avec négation : présentation formelle. Rap. de Recherche LIFO 87-3, Université d'Orléans, June 1987.

[DF 87b] Deransart P., Ferrand G. : An Operational formal definition of PROLOG. 4th Symposium on Logic Programming, San Francisco, September 1987.

[DF 88a] Deransart P., Ferrand G. : Introduction à la Méthodologie de Programmation en Logique. NOVIA, INRIA Report, January 1988.

[DF 88b] Deransart P., Ferrand G. : Logic Programming : Methodology and Teaching. K. Fuchi, L. Kott Editors, French Japan Symposium, North Holland, pp 133-147, August 1988.

[DF 88c] Deransart P., Ferrand G. : On the Semantics of Logic Programming with Negation. RR 88-1, LIFO, University of Orléans, January 1988.

[DM 85] Deransart P., Maluszynski J. : Relating Logic Programs and Attribute Grammars. Journal of Logic Programming 1985, 2, pp 119-155. INRIA RR 393, April 1985.

[DM 88] Deransart P., Maluszynski J. : A grammatical View of Logic Programming. PLILP'88, Orléans, France, May 16-18, 1988.

[DrM 87] Drabent W., Maluszynski J. : Inductive Assertion Method for Logic Programs. CFLP 87, Pisa, Italy, March 23-27, 1987 (also : Proving-Run-Time Properties of Logic Programs. University of Linköping. IDA R-86-23 Logprog, July 1986).

[Fer 86] Ferrand G. : A reconstruction of Logic Programming with Negation. Publication LIFO n° 86-5, University of Orléans, December 1986.

[Fer 87] Ferrand G. : Error diagnosis in Logic Programming, an adaptation of E.Y. Shapiro's method. Journal of Logic Programming 4, pp 177-198, 1987.

[Fri 88] Fribourg L. : Equivalence-Preserving Transformations of Inductive Properties of Prolog Programs. ICLP'88, Seattle, August 1988.

[Hog 84] Hogger C. J. : Introduction to Logic Programming. APIC Studies in Data Processing n° 21, Academic Press, 1984.

[Kow 74] Kowalski R. A. : Predicate Logic as a Programming Language. IFIP 74, North Holland, pp 569-574, 1974.

[Kow 79] Kowalski R. A. : Algorithm = Logic + Control. CACM, 22, 7, July 1979.

[KS 86] Kanamori T., Seki H. : Verification of Prolog Programs using an Extension of Execution. In (Shapiro E., Ed) 3rd ICLP, LNCS 225, pp 475-489, Springer Verlag, 1986.

[Kun 87] Kunen K. : Negation in Logic Programming. Journal of Logic Programming 4, pp 289-308, 1987.

[Llo 87] Lloyd J. W. : Foundations of Logic Programming (2nd ed). Springer Verlag, 1987.

[PP 88] Przymunsinska H., Przymunsinski T. : Weakly Perfect Model Semantics for Logic Programs, pp 1106-1122, ICLP'88, Seattle, 1988.

[Rei 78] Reiter R. : On closed world data bases. Gallaire H. and Minker J. (eds), Logic and Data Bases, Plenum, New York, 1978.

[Rob 68] Robinson J. A. : A machine Oriented Logic Based on the Resolution Principle. JACM, V12 n° 1, 1968.

[She 85] Sheperdson J. C. : Negation as failure I, II. Journal of Logic Programming 1, pp 1-48, 1984, 3, pp 185-202, 1985.

[SS 86] Sterling L., Shapiro E.Y. : The Art of Prolog. MIT Press, 1986.

[Tär 86] Tärlund S. A. : Logic programming - from a Logic Point of view. Symp. on Logic Programming, Salt Lake City, pp 96-103, 1986.

[VEK 76] Van Emden M. H., Kowalski R. A. : The semantics of predicate logic as a programming language. JACM 23 (4), pp 733-742, 1976.

[Wal 87] Wallace M. : Negation in deductive databases. 4th Symposium on Logic Programming, San Francisco, September 1987.

[WML 84] Wolfram D.A. , Maher M.J., Lassez J.L. : A Unified Treatment of Resolution Strategies for Logic Programs. Proceedings of the 2nd ILPC, Uppsala, pp 263-276, July 2-6, 1984.0000